Einführung in PROLOG IV

Listen

Beispiele für Listen in PROLOG:

1. **[mia, vincent, jules, yolanda]**
2. **[mia, robber(honey_bunny), X, 2, mia]**
3. **[]**
4. **[mia, [vincent, jules], [butch, girlfriend(butch)]]**
5. **[[], dead(zed), [2, [b, chopper]], [], Z, [2, [b, chopper]]]]**

Wir erkennen draus:

- Die Elemente stehen, von Kommas getrennt in eckigen Klammer. Die Länge der ersten Liste (Anzahl der Elemente) beträgt 4.
- Alle Arten von PROLOG-Objekten können Listenelemente sein, und beliebig oft in einer Liste auftauchen.
- Es gibt die leere Liste (Länge 0)
- Listen können wieder Elemente anderer Listen sein. Die Länge der 4. Liste ist **3**!

Eine nichtleere Liste besteht immer aus zwei Teilen, dem *Kopf* und dem *Rumpf.*
Im ersten Beispiel ist der Kopf *mia*, der Rumpf die Liste *[vincent, jules, yolanda]*.
Mit dem PROLOG-Operator | können wir die Listen mithilfe von matching in Kopf und Rumpf auftrennen:

?- [X|Y] = [mia, vincent, jules, yolanda].

X= mia
Y=[vincent, jules, yolanda]
yes

Oder die ersten zwei Elemente extrahieren :

?- [X,Y | W] = [[], dead(zed), [2, [b, chopper]], [], Z].

X= []
Y= dead(zed)
W= [2, [b, chopper]], [], Z]
Z= Z
yes

(PROLOG informiert uns auch, dass Z noch nicht gebunden wurde)
Wenn uns nur das zweite und vierte Element interessieren:

?- [_, X, _, Y | _] = [[], dead(zed), [2, [b, chopper]], [], Z].

X= dead(zed)
Y= []
Z= Z
yes

Member

Schreiben wir doch einmal ein kleines Programm:

```
member(X, [X | R]).
member(X, [K | R]):- member(X, R).
```

Mit dem Prädikat member können wir also überprüfen, ob X Teil einer Liste ist.

```
?- member( jules, [mia, vincent, jules, yolanda]).

yes
```

Oder eine noch nützlichere Anwendung:

```
?- member(X, [mia, vincent, jules, yolanda]).

X= mia ;
X= vincent ;
X= jules;
X= yolanda;

no
```

Es bietet sich an, member noch etwas eindeutiger zu formulieren:

```
member(X, [X | _]).
member(X, [_ | R]):- member(X, R).
```

Beispiel für rekursives Bearbeiten von Listen

Member arbeitet sich also rekursiv durch einen Liste, indem es etwas mit dem Kopf macht, und dann rekursiv mit dem Rumpf weiterarbeitet. Das ist das klassischste Vorgehen zur Listenbearbeitung bei PROLOG. Daher betrachten wir noch ein Beispiel.
Nehmen wir an, wir haben eine Liste von **a**s und eine von **b**s und benötigen ein Prädikat a2b/2, welches angibt, ob diese Listen die gleiche Länge haben.
Zuerst müssen wir uns Gedanken über den einfachsten Fall machen, also die leere Liste:

```
a2b([], []).
```

Dann können wir rekursiv definieren:

```
a2b( [a | Ra], [b | Rb] ) :- a2b( Ra, Rb).
```

Was passiert bei der Querie:

```
?- a2b(X, [b, b, b]).
```

Was bei:

```
?- a2b(X, Y).
```

Arithmetik

Betrachten wir die PROLOG-eigenen Methoden zur Bearbeitung von ganzen Zahlen :

?- 8 is 6+2.	oder auch	**?- X is 9+2.**
Yes		**X=11**
?- 12 is 6*2.		**?- X is 3*5.**
yes		**X=15**
?- -2 is 6-8.		**?- X is 8-4.**
yes		**X=4**
?- 3 is 6/2.		**?- X is 24/3.**
yes		**X=8**
?- 1 is mod(7,2).		**?-X is mod(15,4).**
yes		**X=3**

Beachte, dass:

?- X = 3+5.

X= 3+5
yes

ergibt.

Arithmetik und Listen

Wie lang ist eine Liste? Hier eine rekursive Definition:

1. Die leere Liste hat die Länge 0.
2. Eine nichtleere Liste hat die Länge 1 + *len(R),* wobei *len(R)* die Länge ihres Rumpfes ist.

Diese Definition liefert uns sofort:

len([], 0).
len([_ | R], N) :- len(R, X), N is X+1.

Dieses Programm funktioniert und ist einfach zu verstehen, aber es gibt noch eine andere Methode, die Länge einer Liste zu ermitteln. Wir benutzen dabei die PROLOG-Technik von *Akkumulatoren*, die sehr wichtig ist.
Es geht darum, Zwischenergebnisse in Variablen zu speichern.
Wir definieren uns akkLen/3 mit folgenden Argumenten:

a**ccLeng(List, Acc, Length)**

List und Length sin klar, was ist aber mit Acc? Hier speichern wir die Zwischenwerte der Länge. Am Anfang ist es also 0, und wären wir rekursiv die Liste abarbeiten, zählen wir immer 1 dazu, wenn wir ein Kopfelement finden, bis wir auf die leere Liste stossen, dann entspricht Acc der Länge:

a**ccLeng([_ | R], A, L):- Anew is A+1, accLeng(R, Anew, L).**
accLeng([], A, A). *(Abbruch muss nach vorne im Programm)*

Jetzt fehlt nur noch ein Prädikat, das arcLeng aufruft und Acc mit 0 initialisiert:

leng(List, Length):- accLeng(List, 0, Length).

Jetzt können wir also Queries aufrufen in der Form:

?- leng([a, b, c, d, e [a, b], g], X).

Vergleichen von ganzen Zahlen

Arithmetisches Beispiel	**PROLOG Schreibweise**
x < y	X < Y
x ≤ y	X =< Y
x = y	X = := Y
x ≠ y	X =\= Y

(ebenso bei >)

Beachten Sie den Unterschied zwischen = := und =

?- 4=4.
yes

?- 2+2=4.
no

?- 2+2 =:= 4.
yes

Wann immer wir diese Operatoren benutzen, müssen wir sicherstellen, dass die Variablen instanziiert sind. Alle folgenden Queries würden z.B. bei *zu instantiation errors* führen:

?- X < 3.

?- 3 > Y.

?- X =:= X.

Schauen wir uns ein Beispiel an. Das folgende Prädikat erhält eine Liste mit ganzen Zahlen als erstes Argument und gibt die größte Zahl als letztes Argument zurück:

```
accMax( [K | R], A, Max):-
                K > A,
                accMax( R, K, Max).
accMax( [K | R], A, Max):-
                K =< A,
                accMax( R, A, Max).
accMax( [], A, A). (Abbruch muss nach vorne im Programm)
```

Jetzt fehlt noch ein aufrufendes Prädikat, diesmal initialisieren wir Acc mit dem Kopfelement der Liste (warum?):

```
max(List, Max):-
                List = [ K | _ ],
                accMax( List, K, Max).
```

Übungen (Listen)

1. Was antwortet PROLOG auf die folgenden Queries?

1. `[a,b,c,d] = [a,[b,c,d]].`
2. `[a,b,c,d] = [a|[b,c,d]].`
3. `[a,b,c,d] = [a,b,[c,d]].`
4. `[a,b,c,d] = [a,b|[c,d]].`
5. `[a,b,c,d] = [a,b,c,[d]].`
6. `[a,b,c,d] = [a,b,c|[d]].`
7. `[a,b,c,d] = [a,b,c,d,[]].`
8. `[a,b,c,d] = [a,b,c,d|[]].`
9. `[] = _.`
10. `[] = [_].`
11. `[] = [_|[]].`

2.Betrachten wir folgende knowledge base:

```
tran(eins,one).
tran(zwei,two).
tran(drei,three).
tran(vier,four).
tran(fuenf,five).
tran(sechs,six).
tran(sieben,seven).
tran(acht,eight).
tran(neun,nine).
```

Schreiben Sie ein Prädikat `listtran(G,E)` welches eine Liste von englischen Zahlen in eine Liste von Deutschen Zahlen übersetzt. Zum Beispiel:

```
listtran([eins,neun,zwei],X).
```

soll ergeben:

```
X = [one,nine,two].
```

Oder anders herum sollte die Querie:

```
listtran(X,[one,seven,six,two]).
```

Zur Folge haben:

```
X = [eins,sieben,sechs,zwei].
```

Hinweis: Überlegen Sie zuerst, wie leere Listen zu übersetzen sind, das ist dann wieder die Basisregel, der rekursive Teil folgt wie oben.

3. Schreiben Sie ein Prädikat `combine1`, welches drei Listen als Argumente hat und die Elemente der ersten beiden Liste in die Dritte folgendermaßen kombiniert:

```
combine1([a,b,c],[1,2,3],X).

X = [a,1,b,2,c,3]

combine1([foo,bar,yip,yup],[glub,glab,glib,glob],Result).

Result = [foo,glub,bar,glab,yip,glib,yup,glob]
```

4.Jetzt ein dreistelliges Prädikat combine2 ,welches folgendermaßen kombiniert:

```
combine1([a,b,c],[1,2,3],X).

X = [[a,1],[b,2],[c,3]]

combine1([foo,bar,yip,yup],[glub,glab,glib,glob],Result).

Result = [[foo,glub],[bar,glab],[yip,glib],[yup,glob]]
```

5.Und nun noch ein combine2 mit folgender Wirkung:

```
combine1([a,b,c],[1,2,3],X).

X = [join(a,1),join(b,2),join(c,3)]

combine1([foo,bar,yip,yup],[glub,glab,glib,glob],Result).

Result = [join(foo,glub),join(bar,glab),join(yip,glib),join(yup,glob)]
```

Die drei combine2 sind sich sehr ähnlich. Etwas anspruchsvoller ist:

6. Ein Prädikat mysubset/2 ,dass zwei Listen als Argumente hat und testet, ob die erste Liste Teilmenge der zweiten ist.

Übungen Aritmetik

7. Verändern Sie das Prädikat accMax ein wenig um zu accMin, das das Minimum einer Zahlenliste herausgibt.

8. Schreiben Sie ein Prädikat skalarMult, das eine skalare Multiplikation eines Vektors mit einem Skalar ausführt. Im Beispiel:

?- skalarMult(3, [2, 4,1], Ergebnis).

Ergebnis = [6, 12, 3]
yes

9. Und noch ein Prädikat skalarProd für das Skalarprodukt:

?- skalarProd([2, 2, 3], [1, 2, 3], Ergebnis).

Ergebnis = 15
yes

beides für beliebig-dimensionale Vektoren.